Partial Order Reductions using Compositional Confluence Detection

Frédéric Lang, Radu Mateescu

Partial Order Reductions using Compositional Confluence Detection

Frédéric Lang and Radu Mateescu

Vasy project-team, Inria Grenoble Rhône-Alpes/Lig, Montbonnot, France
{Frederic.Lang,Radu.Mateescu}@inria.fr

Abstract. Explicit state methods have proven useful in verifying safety-critical systems containing concurrent processes that run asynchronously and communicate. Such methods consist of inspecting the states and transitions of a graph representation of the system. Their main limitation is state explosion, which happens when the graph is too large to be stored in the available computer memory. Several techniques can be used to palliate state explosion, such as on-the-fly verification, compositional verification, and partial order reductions. In this paper, we propose a new technique of partial order reductions based on compositional confluence detection (Ccd), which can be combined with the techniques mentioned above. Ccd is based upon a generalization of the notion of confluence defined by Milner and exploits the fact that synchronizing transitions that are confluent in the individual processes yield a confluent transition in the system graph. It thus consists of analysing the transitions of the individual process graphs and the synchronization structure to identify such confluent transitions compositionally. Under some additional conditions, the confluent transitions can be given priority over the other transitions, thus enabling graph reductions. We propose two such additional conditions: one ensuring that the generated graph is equivalent to the original system graph modulo branching bisimulation, and one ensuring that the generated graph contains the same deadlock states as the original system graph. We also describe how Ccd-based reductions were implemented in the Cadp toolbox, and present examples and a case study in which adding Ccd improves reductions with respect to compositional verification and other partial order reductions.

1 Introduction

This paper deals with systems, hereafter called *asynchronous systems*, which can be modeled by a composition of individual processes that execute in parallel at independent speeds and communicate. Asynchronous systems can be found in many application domains, such as communication protocols, embedded software, hardware architectures, distributed systems, etc.

Industrial asynchronous systems are often subject to strong constraints in terms of development cost and/or reliability. A way to address these constraints is

to use methods allowing the identification of bugs as early as possible in the development cycle. Explicit state verification is such a method, and consists of verifying properties by systematic exploration of the states and transitions of an abstract model of the system.

Although appropriate for verifying asynchronous systems, explicit state verification may be limited by the combinatorial explosion of the number of states and transitions (called *state explosion*). Among the numerous techniques that have been proposed to palliate state explosion, the following have proved to be effective:

- *On-the-fly verification* (see e.g., [9, 8, 21, 31, 29]) consists of enumerating the states and transitions in an order determined by a property of interest, thus enabling one to find property violations before the whole system graph has been generated.
- *Compositional verification* (see e.g., [7, 28, 41, 44, 46, 6, 38, 16, 24, 39, 14, 11]) consists of replacing individual processes by property-preserving abstractions of limited size.
- *Partial order reductions* (see e.g., [15, 42, 35, 20, 43, 36, 37, 19, 33, 3, 34]) consist of choosing not to explore interleavings of actions that are not relevant with respect to either the properties or the graph equivalence of interest.

Regarding partial order reductions, two lines of work coexist. The first addresses the identification of a subset called *persistent* [15] (or *ample* [35], or *stubborn* [42], see [36] for a survey[1]) of the operations that define the transitions of the system, such that all operations outside this subset are independent of all operations inside this subset. This allows the operations outside the persistent subset to be ignored in the current state. Depending on additional conditions, persistent subsets may preserve various classes of properties (e.g., deadlocks, LTL-X, CTL-X, etc.) and/or graph equivalence relations (e.g., branching equivalence [45], weak trace equivalence [5], etc). Other methods based on the identification of independent transitions, such as *sleep sets* [15], can be combined with persistent sets to obtain more reductions.

The second line of work addresses the detection of particular non-observable transitions (non-observable transitions are also called τ-transitions) that satisfy the property of confluence [32, 20, 19, 47, 2, 3, 34], using either symbolic or explicit-state techniques. Such transitions can be given priority over the rest of the transitions of the system, thus avoiding exploration of useless states and transitions while preserving branching (and observational) equivalence. Among the symbolic detection techniques, the proof-theoretic technique of [3] statically generates a formula encoding the confluence condition from a μCRL program, and then solves it using a separate theorem prover. Among the explicit-state techniques, the global technique of [19] computes the maximal set of strongly confluent τ-transitions and reduces the graph with respect to this set. A local

[1] In this paper, the term *persistent* will refer equally to persistent, ample, or stubborn.

technique was proposed in [2], which computes on-the-fly a representation map associating a single state to each connected subgraph of confluent τ-transitions. Another technique was proposed in [34], which reformulates the detection as the resolution of a BES (*Boolean Equation System*) and prioritizes confluent τ-transitions in the individual processes before composing them, using the fact that branching equivalence is a congruence for the parallel composition of processes. Compared to persistent subset methods, whose practical effectiveness depends on the accuracy of identifying independent operations (by analyzing the system description), confluence detection methods are able to detect all confluent transitions (by exploring the system graph), potentially leading to better reductions.

In this paper, we present a new compositional partial order reduction method for systems described as networks of communicating automata. This method, named CCD (*Compositional Confluence Detection*), exploits the confluence of individual process transitions that are not necessarily labeled by τ and thus cannot be prioritized in the individual processes. CCD relies on the fact that synchronizing such transitions always yields a confluent transition in the graph of the composition. As an immediate consequence, if the latter transition is labeled by τ (i.e., hidden after synchronization), then giving it priority preserves branching equivalence. We also describe conditions to ensure that even transitions that are not labeled by τ can be prioritized, while still preserving the deadlocks of the system.

The aim of CCD is to use compositionality to detect confluence more efficiently than explicit-state techniques applied directly to the graph of the composition, the counterpart being that not all confluent transitions are necessarily detected (as in persistent subset methods). Nevertheless, CCD and persistent subset methods are orthogonal, meaning that neither method applied individually performs better than both methods applied together. Thus, CCD can be freely added in order to improve the reductions achieved by persistent subset methods. Moreover, the definition of confluent transitions is language-independent (i.e., it does not rely upon the description language — in our case EXP.OPEN 2.0 [25] — but only upon the system graph), making CCD suitable for networks of communicating automata produced from any description language equipped with interleaving semantics.

CCD was implemented in the CADP toolbox [12] and more particularly in the existing EXP.OPEN 2.0 tool for compositional verification, which provides on-the-fly verification of compositions of processes. A new procedure was developed, which searches and annotates the confluent (or strictly confluent) transitions of a graph, using a BES to encode the confluence property. This procedure is invoked on the individual processes so that EXP.OPEN 2.0 can then generate a reduced graph for the composition, possibly combined with already available persistent subset methods.

Experimental results show that adding CCD may improve reductions with respect to compositional verification and persistent subset methods.

Paper outline. Section 2 gives preliminary definitions and theorems. Section 3 formally presents the semantic model that we use to represent asynchronous systems. Section 4 presents the main result of the paper. Section 5 describes how the CCD technique is implemented in the CADP toolbox. Section 6 presents several experimental results. Section 7 reports about the application of CCD in an industrial case-study. Finally, Section 8 gives concluding remarks.

2 Preliminaries

We consider the standard LTS (*Labeled Transition System*) semantic model [32], which is a graph consisting of a set of *states*, an *initial state*, and a set of *transitions* between states, each transition being labeled by an action of the system.

Definition 1 (Labeled Transition System). Let $\mathcal{A}$ be a set of symbols called *labels*, which contains a special symbol τ, called the *unobservable label*. An LTS is a quadruple $(Q, A, \rightarrow, q_0)$, where Q is the set of *states*, $A \subseteq \mathcal{A}$ is the set of *labels*, $\rightarrow \subseteq Q \times A \times Q$ is the *transition relation*, and $q_0 \in Q$ is the *initial state* of the LTS. As usual, we may write $q_1 \xrightarrow{a} q_2$ instead of $(q_1, a, q_2) \in \rightarrow$. Any sequence of the form $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \ldots q_n \xrightarrow{a_n} q_{n+1}$ is called a *path of length* n from q_1 to q_{n+1} ($n \geq 0$). We write $q_1 \longrightarrow^n q_{n+1}$ if there exists such a path. The transition relation is acyclic if every path from a state to itself has length 0. □

Branching equivalence [45] is a weak bisimulation relation between states of an LTS that removes some τ-transitions while preserving the branching structure of the LTS. Therefore, branching equivalence is of interest when verifying branching-time temporal logic properties that concern only observable labels.

Definition 2 (Branching equivalence [45]). As usual, we write $\xrightarrow{\tau*}$ the reflexive and transitive closure of $\xrightarrow{\tau}$. Two states $q_1, q_2 \in Q$ are *branching equivalent* if and only if there exists a relation $R \subseteq Q \times Q$ such that $R(q_1, q_2)$ and (1) for each transition $q_1 \xrightarrow{a} q_1'$, either $a = \tau$ and $R(q_1', q_2)$ or there is a path $q_2 \xrightarrow{\tau*} q_2' \xrightarrow{a} q_2''$ such that $R(q_1, q_2')$ and $R(q_1', q_2'')$, and (2) for each transition $q_2 \xrightarrow{a} q_2'$, either $a = \tau$ and $R(q_1, q_2')$ or there is a path $q_1 \xrightarrow{\tau*} q_1' \xrightarrow{a} q_1''$ such that $R(q_1', q_2)$ and $R(q_1'', q_2')$. □

The following definition of *strong confluence* is a synthesis of the definitions of *confluence* by Milner [32], which is a property of processes, and *partial strong τ-confluence* by Groote and van de Pol [19], which is a property of τ-transitions. We thus generalize Groote and van de Pol's definition to transitions labeled by arbitrary symbols, as was the case of Milner's original definition. In addition, we distinguish between the property of strong confluence, and a slightly more constrained property, named *strict* strong confluence.

Definition 3 (Strong confluence). Let $(Q, A, \rightarrow, q_0)$ be an LTS and $T \subseteq \rightarrow$. We write $q \xrightarrow{a}_T q'$ if $(q, a, q') \in T$. We write $q \xrightarrow{\overline{a}} q'$ if either $q \xrightarrow{a} q'$ or $q = q'$

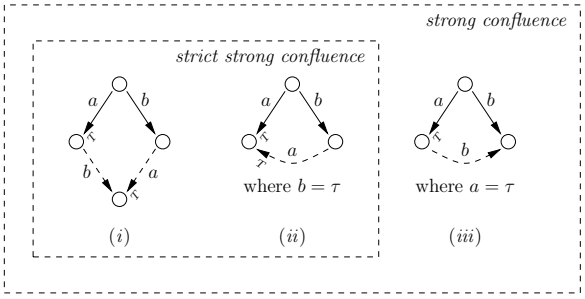


Fig. 1. Graphical definition of *strong confluence* and *strict strong confluence*

and $a = \tau$, and similarly for $q \xrightarrow{\overline{a}}_T q'$. T is *strongly confluent* if for every pair of distinct transitions $q_1 \xrightarrow{a}_T q_2$ and $q_1 \xrightarrow{b} q_3$, there exists a state q_4 such that $q_3 \xrightarrow{\overline{a}}_T q_4$ and $q_2 \xrightarrow{\overline{b}} q_4$. T is *strictly strongly confluent* if for every pair of distinct transitions $q_1 \xrightarrow{a}_T q_2$ and $q_1 \xrightarrow{b} q_3$, there exists a state q_4 such that $q_3 \xrightarrow{a}_T q_4$ and $q_2 \xrightarrow{\overline{b}} q_4$. A transition is *strongly confluent* (respectively *strictly strongly confluent*) if there exists a strongly confluent set (respectively strictly strongly confluent set) $T \subseteq \rightarrow$ containing that transition. □

Figure 1 gives a graphical picture of strong confluence. Plain arrows denote transitions quantified universally, whereas dotted arrows denote transitions quantified existentially. For strict strong confluence, case (*iii*) is excluded.

Strong τ-confluence is strong confluence of τ-transitions. Weaker notions of τ-confluence have been defined [20, 47], but are out of the scope of this paper. For brevity, we use below the terms confluent and strictly confluent instead of strongly confluent and strictly strongly confluent, respectively.

Prioritization consists of giving priority to some transitions. Definition 4 below generalizes the definition of [19], which was restricted to τ-transitions.

Definition 4 (Prioritization [19]). Let $(Q, A, \rightarrow_1, q_0)$ be an LTS and $T \subseteq \rightarrow_1$. A *prioritization* of $(Q, A, \rightarrow_1, q_0)$ with respect to T is any LTS of the form $(Q, A, \rightarrow_2, q_0)$, where $\rightarrow_2 \subseteq \rightarrow_1$ and for all $q_1, q_2 \in Q, a \in A$, if $q_1 \xrightarrow{a}_1 q_2$ then (1) $q_1 \xrightarrow{a}_2 q_2$ or (2) there exists $q_3 \in Q$ and $b \in A$ such that $q_1 \xrightarrow{b}_2 q_3 \in T$. □

In [19], Groote and van de Pol proved that branching bisimulation is preserved by prioritization of τ-confluent transitions, provided the LTS does not contain cycles of τ-transitions. Theorem 1 below relaxes this constraint by only requiring that the set of prioritized τ-confluent transitions does not contain cycles (which is similar to the cycle-closing condition for ample sets [35]).

Theorem 1. Let $(Q, A, \rightarrow, q_0)$ be an LTS and $T \subseteq \rightarrow$ such that T is acyclic and contains only τ-confluent transitions. Any prioritization of $(Q, A, \rightarrow, q_0)$ with respect to T yields an LTS that is branching equivalent to $(Q, A, \rightarrow, q_0)$. □

Theorem 2 below states that deadlock states can always be reached without following transitions that are in choice with strictly confluent transitions. This allows prioritization of strictly confluent transitions, while ensuring that at least one (minimal) diagnostic path can be found for each deadlock state. The detailed proof can be found in [27].

Theorem 2. Let $(Q, A, \rightarrow, q_0)$ be an LTS, $T \subseteq \rightarrow$ a strictly confluent set of transitions, and $q_\delta \in Q$ be a deadlock state. If $q_1 \longrightarrow^n q_\delta$ and $q_1 \xrightarrow{a}_T q_2$, then $q_2 \longrightarrow^m q_\delta$ with $m < n$. □

Therefore, any prioritization of $(Q, A, \rightarrow, q_0)$ with respect to T yields an LTS that has the same deadlock states as $(Q, A, \rightarrow, q_0)$.

Note. Theorem 2 is not true for non-strict confluence, as illustrated by the LTS consisting of the transition $q_1 \xrightarrow{a} q_\delta$ and the (non-strictly) confluent transition $q_1 \xrightarrow{\tau} q_1$.

3 Networks of LTSs

This section introduces networks of LTSs [25, 26], a concurrent model close to MEC [1] and FC2 [4], which consists of a set of LTSs composed in parallel and synchronizing following general synchronization rules.

Definition 5 (Vector). A *vector* of *length* n over a set T is an element of T^n. Let $\boldsymbol{v}$, also written $(v_1, \ldots, v_n)$, be a vector of length n. The elements of $1..n$ are called the *indices* of $\boldsymbol{v}$. For each $i \in 1..n$, $\boldsymbol{v}[i]$ denotes the i^{th} element v_i of $\boldsymbol{v}$. □

Definition 6 (Network of LTSs). Let $\bullet \notin \mathcal{A}$ be a special symbol denoting *inaction*. A *synchronization vector* is a vector over $\mathcal{A} \cup \{\bullet\}$. Let $\boldsymbol{t}$ be a synchronization vector of length n. The *active components* of $\boldsymbol{t}$, written $\text{act}(\boldsymbol{t})$, are defined as the set $\{i \in 1..n \mid \boldsymbol{t}[i] \neq \bullet\}$. The *inactive components* of $\boldsymbol{t}$, written $\text{inact}(\boldsymbol{t})$, are defined as the set $1..n \setminus \text{act}(\boldsymbol{t})$. A *synchronization rule* of length n is a pair $(\boldsymbol{t}, a)$, where $\boldsymbol{t}$ is a synchronization vector of length n and $a \in \mathcal{A}$. The elements $\boldsymbol{t}$ and a are called respectively the *left-* and *right-hand sides* of the synchronization rule. A *network of* LTSs N of length n is a pair $(\boldsymbol{S}, V)$ where $\boldsymbol{S}$ is a vector of length n over LTSs and V is a set of synchronization rules of length n. □

In the sequel, we may use the term network instead of network of LTSs. A network $(\boldsymbol{S}, V)$ therefore denotes a product of LTSs, where each rule expresses

a constraint on the vector of LTSs $\boldsymbol{S}$. In a given state of the product, each rule $(\boldsymbol{t}, a) \in V$ yields a transition labeled by a under the condition that, assuming $\mathrm{act}(\boldsymbol{t}) = \{i_0, \ldots, i_m\}$, the LTSs $\boldsymbol{S}[i_0], \ldots, \boldsymbol{S}[i_m]$ may synchronize altogether on transitions labeled respectively by $\boldsymbol{t}[i_0], \ldots, \boldsymbol{t}[i_m]$. This is described formally by the following definition.

Definition 7 (Network semantics). Let N be a network of length n defined as a couple $(\boldsymbol{S}, V)$ and for each $i \in 1..n$, let $\boldsymbol{S}[i]$ be the LTS $(Q_i, A_i, \rightarrow_i, q_{0i})$. The *semantics* of N, written $lts(N)$ or $lts(\boldsymbol{S}, V)$, is an LTS $(Q, A, \rightarrow, \boldsymbol{q}_0)$ where $Q \subseteq Q_1 \times \ldots \times Q_n$, $\boldsymbol{q}_0 = (q_{01}, \ldots, q_{0n})$ and $A = \{a \mid (\boldsymbol{t}, a) \in V\}$. Given a synchronization rule $(\boldsymbol{t}, a) \in V$ and a state $\boldsymbol{q} \in Q_1 \times \ldots \times Q_n$, we define the *successors* of $\boldsymbol{q}$ by rule $(\boldsymbol{t}, a)$, written $\mathrm{succ}(\boldsymbol{q}, (\boldsymbol{t}, a))$, as follows:

$$\mathrm{succ}(\boldsymbol{q}, (\boldsymbol{t}, a)) = \{\boldsymbol{q}' \in Q_1 \times \ldots \times Q_n \mid (\forall i \in \mathrm{act}(\boldsymbol{t}))\ \boldsymbol{q}[i] \xrightarrow{\boldsymbol{t}[i]}_i \boldsymbol{q}'[i] \wedge (\forall i \in \mathrm{inact}(\boldsymbol{t}))\ \boldsymbol{q}[i] = \boldsymbol{q}'[i]\}$$

The state set Q and the transition relation $\rightarrow$ of $lts(N)$ are the smallest set and the smallest relation such that $\boldsymbol{q}_0 \in Q$ and:
$q \in Q \wedge (\boldsymbol{t}, a) \in V \wedge \boldsymbol{q}' \in \mathrm{succ}(\boldsymbol{q}, (\boldsymbol{t}, a)) \Rightarrow \boldsymbol{q}' \in Q \wedge \boldsymbol{q} \xrightarrow{a} \boldsymbol{q}'$. □

Synchronization rules must obey the following admissibility condition, which forbids cutting, synchronization and renaming of the τ transitions present in the individual LTSs. This is suitable for a process algebraic framework, most parallel composition, hiding, renaming, and cutting operators of which can be translated into rules obeying these conditions. This also ensures that weak trace equivalence and stronger relations (e.g., safety, observational, branching, and strong equivalences) are congruences for synchronization rules [25].

Definition 8 (Network admissibility). The network $(\boldsymbol{S}, V)$ is *admissible* if for each q, q', i such that $q \xrightarrow{\tau}_i q'$ there exists a rule $(\boldsymbol{t}_i, \tau) \in V$ where $\boldsymbol{t}_i[i] = \tau$, $(\forall j \neq i)\ \boldsymbol{t}_i[j] = \bullet$, and $(\forall (t, a) \in V \setminus \{(\boldsymbol{t}_i, \tau)\})\ t[i] \neq \tau$. Below, every network will be assumed to be admissible. □

Example 1. We consider the simple network of LTSs consisting of the vector of LTSs $(Sender_1, Bag, Sender_2)$ depicted in Figure 2 (the topmost node being the initial state of each LTS), and of the following four synchronization rules: $((s_1, s_1, \bullet), \tau),\ ((\bullet, s_2, s_2), \tau),\ ((\bullet, r_1, \bullet), r_1),\ ((\bullet, r_2, \bullet), r_2)$.

This network represents two processes $Sender_1$ and $Sender_2$, which send their respective messages s_1 and s_2 via a communication buffer that contains one place for each sender and uses a *bag* policy (received messages can be delivered in any order). Every transition in the individual LTSs of this network is strictly confluent. The LTS (i) depicted in Figure 3, page 9, represents the semantics of this network.

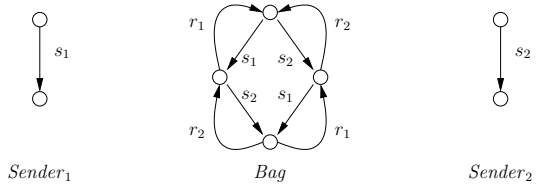


Fig. 2. Individual LTSs of the network defined in Example 1

4 Compositional Confluence Detection

Although prioritizing confluent transitions yields LTS reductions, finding confluent transitions in large LTSs such as those obtained by parallel composition of smaller LTSs can be quite expensive in practice. Instead, the aim of CCD is to infer confluence in the large LTS from the (much cheaper to find) confluence present in the smaller LTSs that are composed.

Definition 9. Let $(\boldsymbol{S}, V)$ be a network, $(\boldsymbol{t}, a) \in V$, and $\boldsymbol{q}$, $\boldsymbol{q'}$ be states of $lts(\boldsymbol{S}, V)$. We write $\text{all\_conf}(\boldsymbol{q}, (\boldsymbol{t}, a), \boldsymbol{q'})$ for the predicate "$\boldsymbol{q'} \in \text{succ}(\boldsymbol{q}, (\boldsymbol{t}, a)) \wedge (\forall i \in \text{act}(\boldsymbol{t}))\ \boldsymbol{q}[i] \xrightarrow{\boldsymbol{t}[i]}_i \boldsymbol{q'}[i]$ is confluent". We write all_conf_strict for the same predicate, where "*strictly confluent*" replaces "*confluent*". □

Theorem 3 below presents the main result of this paper: synchronizations involving only confluent (resp. strictly confluent) transitions in the individual LTSs produce confluent (resp. strictly confluent) transitions in the LTS of the network.

Theorem 3 (Compositional confluence detection). Let $(\boldsymbol{S}, V)$ be a network, $(\boldsymbol{t}, a) \in V$, and $\boldsymbol{q}$, $\boldsymbol{q'}$ be states of $lts(\boldsymbol{S}, V)$. (1) If $\text{all\_conf}(\boldsymbol{q}, (\boldsymbol{t}, a), \boldsymbol{q'})$, then $\boldsymbol{q} \xrightarrow{a} \boldsymbol{q'}$ is confluent and (2) if $\text{all\_conf\_strict}(\boldsymbol{q}, (\boldsymbol{t}, a), \boldsymbol{q'})$, then $\boldsymbol{q} \xrightarrow{a} \boldsymbol{q'}$ is strictly confluent. □

The proof [27] consists of showing that the set $\{\boldsymbol{p} \xrightarrow{a} \boldsymbol{p'} \mid \text{all\_conf}(\boldsymbol{p}, (\boldsymbol{t}, a), \boldsymbol{p'})\}$ is indeed a confluent set (and similarly for the strictly confluent case).

We call *deadlock preserving reduction using* CCD a prioritization of transitions obtained from synchronization of strictly confluent transitions (which indeed preserves the deadlocks of the system following Theorems 2 and 3), and *branching preserving reduction using* CCD a prioritization of τ-transitions obtained from synchronization of confluent transitions, provided they are acyclic (which indeed preserves branching bisimulation following Theorems 1 and 3). The major differences between both reductions are thus the following: (1) branching preserving reduction does not require strict confluence; (2) deadlock preserving reduction does not require any acyclicity condition; and (3) deadlock preserving reduction does not require the prioritized transitions to be labeled by τ, which preserves the labels of diagnostic paths leading to deadlock states.

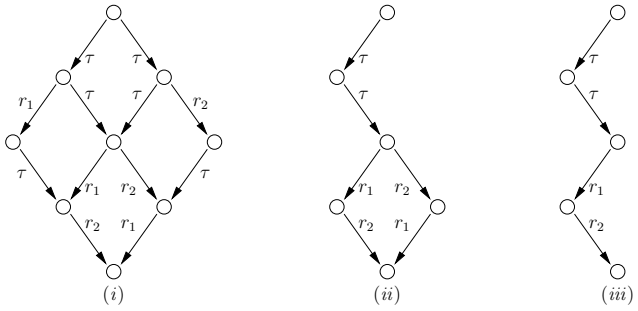


Fig. 3. Three LTSs corresponding to the semantics of the network of Example 1, one generated without CCD (*i*) and two generated using CCD preserving respectively branching equivalence (*ii*) and deadlocks (*iii*)

Example 2. Figure 3 depicts three LTSs corresponding to the network presented in Example 1, page 7. LTS (*i*) corresponds to the semantics of the network, generated without reduction. LTS (*ii*) is the same generated with branching preserving reduction using CCD and thus is branching equivalent to LTS (*i*). LTS (*iii*) is the same generated with deadlock preserving reduction using CCD and thus has the same deadlock state as LTS (*i*).

As persistent subset methods, CCD is able to detect commuting transitions by a local analysis of the network. For persistent subsets, a relation of independence between the transitions enabled in the current state is computed dynamically by inspection of the transitions enabled in the individual LTSs and of their interactions (defined here as synchronization rules). By contrast, CCD performs a static analysis of the individual LTSs to detect which transitions are locally confluent, the dynamic part being limited to checking whether a transition of the network can be obtained by synchronizing only locally confluent transitions.

Branching preserving reduction using CCD does not require detection of all confluent transitions in the individual LTSs of the network, but can be restricted to those active in a synchronization rule of the form $(\boldsymbol{t}, \tau)$. In a network $(\boldsymbol{S}, V)$ of length n, we thus compute for each $i \in 1..n$ a subset $C_i \subseteq A_i$ of labels that contains all labels $\boldsymbol{t}[i] \neq \bullet$ such that there exists $(\boldsymbol{t}, \tau) \in V$. For deadlock preserving reduction, the subset C_i is defined as A_i.

The problem of detecting confluence in the individual LTSs is reformulated in terms of the local resolution of a BES (*Boolean Equation System*), following the scheme we proposed in [34]. Given an LTS $(Q_i, A_i, \rightarrow_i, q_{0i})$ and a subset $C_i \subseteq A_i$ of labels, the BES encoding the detection of confluent transitions labeled by actions in C_i is defined as follows:

$$\left\{ X_{q_1 a q_2} \stackrel{\nu}{=} \bigwedge_{q_1 \xrightarrow{b}_i q_3} \bigvee_{q_2 \xrightarrow{\overline{b}}_i q_4} \left(\bigvee_{q_3 \xrightarrow{\overline{a}}_i q_4'} (q_4 = q_4' \wedge X_{q_3 a q_4}) \vee (a = \tau \wedge q_3 = q_4) \right) \right\}$$

Each boolean variable $X_{q_1 a q_2}$, where $q_1, q_2 \in Q_i$ and $a \in C_i$, evaluates to true if and only if $q_1 \xrightarrow{a}_i q_2$ is confluent. The BES has maximal fixed point semantics because we seek to determine the maximal set of confluent transitions contained in an LTS. For strict confluence, $\bigvee_{q_3 \xrightarrow{\overline{a}}_i q'_4}$ must be merely replaced by $\bigvee_{q_3 \xrightarrow{a}_i q'_4}$.

The correctness of this encoding [27] is based upon a bijection between the fixed point solutions of the BES and the sets of confluent transitions labeled by actions in C_i; thus, the maximal fixed point solution gives the whole set of such confluent transitions.

5 Implementation

CCD was implemented in CADP[2] (*Construction and Analysis of Distributed Processes*) [12], a toolbox for the design of communication protocols and distributed systems, which offers a wide set of functionalities, ranging from step-by-step simulation to massively-parallel model checking. CADP is designed in a modular way and puts the emphasis on intermediate formats and programming interfaces. CADP provides the BCG compact graph format for storing explicit LTSs and the OPEN/CÆSAR [10] application programming interface for representing and manipulating implicit LTSs in the form of an initial state and a successor state function. The GENERATOR tool converts an OPEN/CÆSAR implicit LTS into an explicit BCG graph. The BCG_MIN tool allows minimization of BCG graphs modulo strong and branching bisimulation.

EXP.OPEN 2.0 (an extension of the previous version EXP.OPEN 1.0 of Bozga, Fernandez, and Mounier) is a compiler into OPEN/CÆSAR implicit LTSs of systems made of BCG graphs composed using synchronization vectors and parallel composition, hiding, renaming, and cutting operators taken from the CCS [32], CSP [39], LOTOS [22], E-LOTOS [23], and μCRL [18] process algebras. As an intermediate step, those systems are translated into the network of LTSs model presented in Definition 6. EXP.OPEN 2.0 has several partial order reduction options that allow standard persistent set methods (generalizations of Ramakrishna and Smolka's method presented in [37]) to be applied on-the-fly, among which **-branching** preserves branching bisimulation, **-ratebranching** preserves stochastic branching bisimulation[3], **-deadpreserving** preserves deadlocks, and **-weaktrace** preserves weak trace equivalence (i.e., observable traces).

We developed in the EXP.OPEN 2.0 tool a new procedure that takes as input a BCG graph, a file that contains a set of labels represented using a list of regular expressions, and a boolean parameter for strictness. For each transition whose label matches one of the regular expressions, this procedure checks whether this transition is confluent (or strictly confluent if the boolean parameter is set to

[2] `http://www.inrialpes.fr/vasy/cadp`

[3] This option is similar to **-branching** and additionally gives priority to τ-transitions over stochastic transitions.

true). The BES encoding the confluence detection problem is solved using a global algorithm similar to those in [30]. This produces as output an LTS in the BCG format, the transition labels of which are prefixed by a special tag indicating confluence when appropriate.

We also added to EXP.OPEN 2.0 a new **-confluence** option, which can only be used in combination with one of the partial order reduction options already available (**-branching**, **-deadpreserving**, **-ratebranching**, **-weaktrace**[4]). In this case, EXP.OPEN 2.0 first computes the labels for which confluence detection is useful, and then calls the above procedure (setting the boolean parameter to true if EXP.OPEN was called with the **-deadpreserving** option) on the individual LTSs, providing these labels as input. Finally, it uses the information collected in the individual LTSs to prioritize the confluent transitions on the fly.

6 Experimental Results

We applied partial order reductions using CCD to several examples. To this aim, we used a 2 GHz, 16 GB RAM, dual core AMD Opteron 64-bit computer running 64-bit Linux. Examples identified by a two digit number xy (01, 10, 11, etc.) correspond to LTS compositions extracted from an official CADP demo available at `ftp://ftp.inrialpes.fr/pub/vasy/demos/demo_`xy. These include telecommunication protocols (01, 10, 11, 18, 20, 27), distributed systems (25, 28, 35, 36, 37), and asynchronous circuits (38). Examples $st(1)$, $st(2)$, and $st(3)$ correspond to process compositions provided to us by the STMICROELECTRONICS company, which uses CADP to verify critical parts of their future-generation multiprocessor systems on chip.

In each example, the individual LTSs were first minimized (compositionally) modulo branching bisimulation using BCG_MIN. This already achieves more reduction than the compositional τ-confluence technique presented in [34], since minimization modulo branching bisimulation subsumes τ-confluence reduction. The LTS of their composition was then generated using EXP.OPEN 2.0 and GENERATOR following different strategies: (1) using no partial order reduction at all, (2) using persistent sets, and (3) using both persistent sets and CCD. Figure 4 reports the size (in states/transitions) of the resulting LTS obtained when using option **-branching** (top) or **-deadpreserving** (bottom). The symbol "–" indicates that the number of states and/or transitions is the same as in the column immediately to the left.

These experiments show that CCD may improve the reductions obtained using persistent sets and compositional verification, most particularly in examples 37, 38, $st(1)$, $st(2)$, and $st(3)$. Indeed, in these examples the individual LTSs are themselves obtained by parallel compositions of smaller processes. This tends to generate confluent transitions, which are detected locally by CCD. On the other

[4] Note that branching preserving reduction using CCD also preserves weaker relations such as weak trace equivalence.

Branching preserving reduction

Example	No partial order reduction	Persistent sets	Persistent sets + CCD
01	112/380	−/328	−/−
10	688/2,540	−/2,200	−/−
11	2,995/9,228	−/−	−/9,200
18	129,728/749,312	−/746,880	−/−
20	504,920/5,341,821	−/−	−/5,340,117
25	11,031/34,728	−/−	−/−
27(1)	1,530/5,021	−/−	−/−
27(2)	6,315/22,703	−/−	−/−
28	600/1,925	−/−	−/−
35	156,957/767,211	−/−	−/−
36	23,627/84,707	21/20	−/−
37	22,545/158,318	−/−	541/2,809
38	1,404/3,510	−/3,504	390/591
$st(1)$	6,993/100,566	−/−	−/79,803
$st(2)$	1,109,025/7,448,719	−/−	−/6,163,259
$st(3)$	5,419,575/37,639,782	−/−	5,172,660/24,792,525

Deadlock preserving reduction

Example	No partial order reduction	Persistent sets	Persistent sets + CCD
01	112/380	92/194	−/−
10	688/2,540	568/1,332	−/−
11	2,995/9,228	2,018/4,688	−/4,670
18	129,728/749,312	124,304/689,760	90,248/431,232
20	504,920/5,341,821	481,406/4,193,022	481,397/4,191,555
25	11,031/34,728	6,414/11,625	−/−
27(1)	1,530/5,021	1,524/4,811	−/−
27(2)	6,315/22,703	6,298/22,185	−/−
28	600/1,925	375/902	−/−
35	156,957/767,211	−/−	−/−
36	23,627/84,707	171/170	−/−
37	22,545/158,318	−/−	76/128
38	1,404/3,510	−/3,474	492/673
$st(1)$	6,993/100,566	6,864/96,394	1/1
$st(2)$	1,109,025/7,448,719	−/7,138,844	101,575/346,534
$st(3)$	5,419,575/37,639,782	5,289,255/34,202,947	397,360/1,333,014

Fig. 4. LTS sizes in states/transitions for branching and deadlock preserving reductions

hand, it is not a surprise that neither CCD nor persistent sets methods preserving branching bisimulation reduce examples 25, 27(1), 27(2) and 28, since the resulting LTSs corresponding to these examples contain no confluent transitions.

One might be amazed by the reduction of $st(1)$ to an LTS with only one state and one transition in the deadlock preserving case. The reason is that one LTS of the network has a strictly confluent self looping transition that is independent from the other LTSs. Therefore, the network cannot have a deadlock and is reduced by CCD to this trivial, deadlock-free LTS.

For $st(1)$, $st(2)$, and $st(3)$, we also compared the total time and peak memory needed to generate the product LTS (using EXP.OPEN 2.0/GENERATOR) and then minimize it modulo branching bisimulation (using BCG_MIN), without using any partial order reduction and with persistent sets combined with CCD. This includes time and memory used by the tools EXP.OPEN 2.0, GENERATOR and BCG_MIN. Figure 5 shows that CCD may significantly reduce the total time

	No partial order reduction		Persistent sets + Ccd	
	total time (s)	peak memory (MB)	total time (s)	peak memory (MB)
$st(1)$	0.72	5.6	0.91	5.6
$st(2)$	271	312	287	271
$st(3)$	2,116	1,390	1,588	981

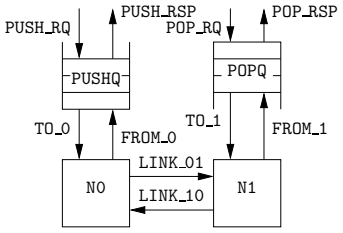
Fig. 5. Resources used to generate and reduce Ltss modulo branching bisimulation

and peak memory (for $st(3)$, 30% and 40%, respectively) needed to generate a minimal Lts.

7 Case study

We present here in more detail the use of Ccd in the context of the Multival project[5], which aims at the formal specification, verification, and performance evaluation of multiprocessor multithreaded architectures developed by Bull, the Cea/Leti, and STMicroelectronics. The case-study below concerns xSTream, a multiprocessor dataflow architecture designed by STMicroelectronics for high performance embedded multimedia streaming applications. In this architecture, computation nodes (e.g., filters) communicate using xSTream queues connected by a NoC (*Network on Chip*) composed of routers connected by direct communication links.

We used as input the network of communicating Ltss produced from a Lotos specification of two xSTream queues connected via a NoC with four routers. The architecture of the system is depicted below, where the components `N0` and `N1` denote the routers involved in the communication between `PUSHQ` and `POPQ`, the behaviour of which incorporates perturbations induced by the other two routers of the NoC.



The Lts of the system can be generated and minimized compositionally using the Svl [11] language of Cadp. The generation was done first with Ccd deactivated, then with Ccd activated. For each case, Figure 6 gives the following information: The "*intermediate*" column indicates the size (in states/transitions) of the intermediate Lts generated by the Exp.Open tool, before minimization

[5] `http://www.inrialpes.fr/vasy/multival`

	without CCD			with CCD		
	intermediate	time (s)	mem. (MB)	intermediate	time (s)	mem. (MB)
itf_POPQ+N1	244,569/1,320,644	18.56	51	179,706/587,187	9.66	26
N0+PUSHQ	22,674/120,222	1.35	17	22,674/86,528	1.12	17
N0+N1+PUSHQ	140,364/828,930	12.62	32	95,208/444,972	6.40	22
NOC4	324,261/2,549,399	11.32	93	310,026/1,073,316	9.77	46

Fig. 6. Performance of LTS generation and minimization with and without CCD

modulo branching bisimulation; The "*time*" and "*mem.*" columns indicate respectively the cumulative time (in seconds) and memory peak (in megabytes) taken by LTS generation (including confluence detection when relevant) and minimization modulo branching bisimulation.

Figure 6 shows that CCD may reduce both the time (the LTSs `"itf_POPQ+NI.bcg"` and `"N0+N1+PUSHQ.bcg"` were generated and minimized twice faster with CCD than without CCD) and memory (`"itf_POPQ+NI.bcg"` and `"NOC4.bcg"` were generated using about half as much memory with CCD as without CCD).

8 Conclusion

CCD (*Compositional Confluence Detection*) is a partial order reduction method that applies to systems of communicating automata. It detects confluent transitions in the product graph, by first detecting the confluent transitions in the individual automata and then analysing their synchronizations. Confluent transitions of the product graph can be given priority over the other transitions, thus yielding graph reductions. We detailed two variants of CCD: one that preserves branching bisimilarity with the product graph, and one that preserves its deadlocks.

CCD was implemented in the CADP toolbox. An encoding of the confluence property using a BES (*Boolean Equation System*) allows the detection of all confluent transitions in an automaton. The existing tool EXP.OPEN 2.0, which supports modeling and verification of systems of communicating automata, was extended to exploit on-the-fly the confluence detected in the individual automata.

CCD can be combined with both compositional verification and other partial order reductions, such as persistent sets. We presented experimental results showing that CCD may significantly reduce both the size of the system graph and the total time and peak memory needed to generate a minimal graph.

As future work, we plan to combine CCD reductions with distributed graph generation [13] in order to further scale up its capabilities. This distribution can be done both at automata level (by launching distributed instances of confluence detection for each automaton in the network or by performing the confluence detection during the distributed generation of each automaton) and at network level (by coupling CCD with the distributed generation of the product graph).

Acknowledgements. We are grateful to W. Serwe (INRIA/VASY) and to E. Lantreibecq (STMicroelectronics) for providing the specifications of the xSTREAM NOC. We also warmly thank the anonymous referees for their useful remarks.

References

1. A. Arnold. MEC: A System for Constructing and Analysing Transition Systems. In *Automatic Verification Methods for Finite State Systems*, *LNCS* 407, 1989.
2. S.C.C. Blom. Partial τ-Confluence for Efficient State Space Generation. Technical Report SEN–R0123, CWI, 2001.
3. S. Blom and J. van de Pol. State Space Reduction by Proving Confluence. In *CAV*, *LNCS* 2404, 2002.
4. A. Bouali, A. Ressouche, V. Roy, R. de Simone. The Fc2Tools set: a Toolset for the Verification of Concurrent Systems. In *CAV*, *LNCS* 1102, 1996.
5. S. D. Brookes, C. A. R. Hoare, A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, 1984.
6. S. C. Cheung and J. Kramer. Enhancing Compositional Reachability Analysis with Context Constraints. In *Foundations of Software Engineering*, 1993.
7. J.-C. Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants*. Thèse de Doctorat, Univ. J. Fourier (Grenoble), 1988.
8. J.-C. Fernandez, C. Jard, T. Jéron, L. Mounier. "On the Fly" Verification of Finite Transition Systems. *FMSD*, 1992.
9. J.-C. Fernandez and L. Mounier. Verifying Bisimulations "On the Fly". In *FDT*, 1990.
10. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *TACAS*, *LNCS* 1384, 1998.
11. H. Garavel and F. Lang. SVL: a Scripting Language for Compositional Verification. In *FORTE*, Kluwer, 2001.
12. H. Garavel, F. Lang, R. Mateescu, W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *CAV*, *LNCS* 4590, 2007.
13. H. Garavel, R. Mateescu, D. Bergamini, A. Curic, N. Descoubes, C. Joubert, I. Smarandache-Sturm, G. Stragier. DISTRIBUTOR and BCG_MERGE: Tools for Distributed Explicit State Space Generation. In *TACAS*, *LNCS* 3920, 2006.
14. D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College, Univ. of London, Dept. of Computer Science, 1999.
15. P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *Computer-Aided Verification*, *DIMACS Series* 3, 1990.
16. S. Graf, B. Steffen, G. Lüttgen. Compositional Minimization of Finite State Systems using Interface Specifications. *FAC*, 8(5):607–616, 1996.
17. S. Graf and B. Steffen. Compositional Minimization of Finite State Systems. In *CAV*, *LNCS* 531, 1990.
18. J. F. Groote and A. Ponse. The Syntax and Semantics of μCRL. In *Algebra of Communicating Processes*, Workshops in Computing Series, 1995.
19. J. F. Groote and J. van de Pol. State Space Reduction using Partial τ-Confluence. In *MFCS*, *LNCS* 1893, 2000.
20. J. F. Groote and M.P.A. Sellink. Confluence for process verification. *TCS*, 170(1–2):47–81, 1996.
21. G. J. Holzmann. On-The-Fly Model Checking. *ACM Comp. Surveys*, 28(4), 1996.

22. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, ISO, 1989.
23. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, ISO, 2001.
24. J.-P. Krimm and L. Mounier. Compositional State Space Generation from LOTOS Programs. In *TACAS*, *LNCS* 1217, 1997.
25. F. Lang. EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. In *IFM*, *LNCS* 3771, 2005.
26. F. Lang. Refined Interfaces for Compositional Verification. In *FORTE*, *LNCS* 4229, 2006.
27. F. Lang and R. Mateescu. Partial Order Reductions using Compositional Confluence Detection. Extended version of FM'09, INRIA, 2009.
28. J. Malhotra, S. A. Smolka, A. Giacalone, R. Shapiro. A Tool for Hierarchical Design and Simulation of Concurrent Systems. In *Specification and Verification of Concurrent Systems*, 1988.
29. R. Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In *TACAS*, *LNCS* 2619, 2003.
30. R. Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *STTT*, 8(1):37–56, 2006.
31. R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *SCP*, 46(3):255–281, 2003.
32. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
33. R. Nalumasu and G. Gopalakrishnan. An Efficient Partial Order Reduction Algorithm with an Alternative Proviso Implementation. *FMSD*, 20(3), 2002.
34. G. Pace, F. Lang, R. Mateescu. Calculating τ-Confluence Compositionally. In *CAV*, *LNCS* 2725, 2003.
35. D. A. Peled. Combining partial order reduction with on-the-fly model-checking. In *CAV*, *LNCS* 818, 1994.
36. D. A. Peled, V. R. Pratt, G. J. Holzmann, editors. *Partial Order Methods in Verification*, *DIMACS Series* 29, 1997.
37. Y. S. Ramakrishna and S. A. Smolka. Partial-Order Reduction in the Weak Modal Mu-Calculus. In *CONCUR*, *LNCS* 1243, 1997.
38. A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, J. B. Scattergood. Hierarchical compression for model-checking CSP *or* how to check 10^{20} dining philosophers for deadlock. In *TACAS*, 1995.
39. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
40. K. Sabnani, A. Lapone, M. Uyar. An Algorithmic Procedure for Checking Safety Properties of Protocols. *IEEE Trans. on Communications*, 37(9):940–948, 1989.
41. K. C. Tai and V. Koppol. Hierarchy-Based Incremental Reachability Analysis of Communication Protocols. In *Network Protocols*, IEEE Press, 1993.
42. A. Valmari. A Stubborn Attack on State Explosion. In *Computer-Aided Verification*, *DIMACS Series* 3, 1990.
43. A. Valmari. Stubborn Set Methods for Process Algebras. In *Partial Order Methods in Verification*, *DIMACS Series* 29, AMS, 1997.
44. A. Valmari. Compositional State Space Generation. In *Advances in Petri Nets*, *LNCS* 674, 1993.
45. R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics. In *IFIP World Computer Congress*, 1989.
46. W. J. Yeh. *Controlling State Explosion in Reachability Analysis*. PhD thesis, Software Engineering Research Center (SERC) Laboratory, Purdue Univ., 1993.
47. M. Ying. Weak confluence and τ-inertness. *TCS*, 238(1–2):465–475, 2000.